Go

Where it's going and why you should pay attention
14 May 2015

Aaron Schlesinger
Sr. Engineer, Iron.io

About Me

- Currently a database & backend systems engineer at Iron.io
- Writing Go for 1.5 years
- Worked on server side and distributed systems for 5 years
- Go is my favorite language. I'm most productive and happiest here

Today

- Why Go is powerful
- Why it's important
- Why it's worth your attention

About Go

A programming language that started at Google. From golang.org (http://golang.org):

```
Go is an open source programming language that makes it easy to build simple,
reliable, and efficient software.
```

Very good choice for:

- Cloud & microservices
- Web servers
- Systems utilities
- Databases
- Monitoring tools

Why Go?

- Efficient runtime
- Simple & powerful primitives
- Extremely productive
- Great tools
- Fun

Simple program

```
package main

import "fmt"

const numIters = 10

func main() {
    for i := 0; i < numIters; i++ {
        fmt.Printf("Hi Gophers! (# %d)\n", i+1)
    }
}
```

Run

Simple (concurrent) program

```
package main

import (
    "fmt"
    "sync"
)

const numIters = 10

func printHello(iterNum int, wg *sync.WaitGroup) {
    defer wg.Done()
    fmt.Printf("Hi Gophers! (# %d)\n", iterNum+1)
}

func main() {
    var wg sync.WaitGroup
    for i := 0; i < numIters; i++ {
        wg.Add(1)
        go printHello(i, &wg)
    }
    wg.Wait()
}
```

Run

Concurrency

It's built into the language. I believe this is the #1 most powerful feature of Go.

- You define concurrent units of execution inside **goroutines** (concurrently executin functions)
- Goroutines can communicate & synchronize using **channels**
- You write your programs as if they do blocking I/O
- Applications use **all** cores

Concurrency details

- Goroutines have dynamically sized stacks
- Runtime multiplexes your Goroutines onto threads
- Runtime automatically context switches for you on I/O, etc...

The current state of Go

Real Go programs are in production at Google, Square, The New York Times, Github, Digital Ocean, Iron.io and many more.

- Active & growing community
- Core team focuses mostly on perf & tools
- Very few new lang features
- "The Go Way"

My predictions

- Go's usage will steadily grow
- More high profile open source projects will be written in Go
- Go will be the best language for cloud computing environments
- Go will be the default stack for many new software businesses

Other programing languages will need to *improve* to:

- Keep pace with growing cloud computing requirements
- Compete with Go

Evidence

I will present evidence for the following:

- Why Go's adoption will grow
- Why Go is the right choice for large scale applications today

Adoption

Why the Go developer base will grow.

1. Buzz

Before a programmer chooses Go, he/she will probably hear about it primarily via blo
posts, articles or word-of-mouth (conferences, meetups, friends).

Many developers of all experience levels are writing and reading about Go today.

1.1. Example

Travis Reeder, co-founder of Iron.io posted about his experience replacing existing software with Go.

blog.iron.io/2013/03/how-we-went-from-30-servers-to-2-go.html (http://blog.iron.io/2013/03/how-we-went-from-30-servers-to-2-go.html)

To date, it is one of Iron's most popular blog posts. It generated a large discussion on Hacker News.

news.ycombinator.com/item?id=5365096 (https://news.ycombinator.com/item?id=5365096)

Similarly, there is significant, growing interest (and debate) around Go in the community at large.

2. First line of code

Anybody can *try* Go by going to play.golang.org (http://play.golang.org).

There are no other requisite steps to execute Go code.

3. Installing Go

Setup for non-trivial development requires the following steps:

- Create and setup your GOPATH
- Download the go binary to your executable path

This process is by far the simplest I have encountered across any language. Go makes an excellent first impression.

4. Toolchain

The toolchain is encapsulated in a single binary. That binary contains all of the tools a first time Go programmer will need to start building and running programs.

It scales to large codebases too. I generally use only 1 additional tool in my everyday development.

5. Dev/test workflow

The language and compiler were designed to compile code quickly and they deliver.

- Fast compilation enables an efficient local edit/test workflow
- Fast compilation enables significant developer efficiency advantages
- Additionally, build/test/deploy pipelines can complete their tasks significantly faste

6. Everyday development

Go enables developers to write better code faster.

- The language is simple (25 keywords) but powerful. Developers focus more on semantics, less on syntax
- Documentation is accessible, centralized and well organized. See godoc.org (http://godoc.
- Industrial grade static checking is built into the standard toolchain
- Go programs are statically linked. Developers run their programs locally with no external dependencies

7. Extensibility

The Go toolchain's feature set will grow primarily because the language is so simple and powerful.

For example:

There's a Go parser built into the standard library. Developers in the community have used this feature to quickly build high quality code generation tools, new static checkers, etc...

8. Packaging

Go includes a simple packaging scheme:

- The toolchain installs a dependency by downloading the code and compiling it into your program
- Packages can be hosted on major source code repositories or any server that follows a simple HTTP request protocol

Go packaging is criticized a lot. The community has built many tools to make it more robust, but its simplicity enables an extremely low friction release process.

Suitability for cloud & microservices environments

Why Go is here to stay.

1. Deployment

The following key features make Go programs very easy to deploy in multiple scenarios

- The Go compiler statically links its output binaries
- Go binaries require no runtime/interpreter on the target. All necessary runtime components are compiled into the binary
- The Go toolchain can cross compile programs. A developer on a Mac OS X machin can compile for a Linux target platform, for example

1.1. Deploying a web application

Statically linked binaries with no external dependencies (except libc if linux target) are much easier to deploy than dynamically linked binaries or JVM applications.

Engineers won't choose Go for this feature, but they appreciate it after they do.

1.2. Deploying a CLI application

Some developers are beginning to use Go for CLI applications as well. Cross compiling and static linking are both extremely important in this scenario.

Go allows CLI developers to ship a single binary without writing detailed install instructions or worrying about dependencies on the user's machine.

2. Concurrency & I/O

Modern web and CLI applications alike often need to do complex I/O.

Web apps, for example, often need to interact with multiple 3rd party APIs which may fail independently.

Go has concurrency & I/O features that enable robust applications:

- Runtime-managed goroutines
- Built in, typed *channels* for cross-goroutine communication and synchronization
- *select* statements for doing complex operations on multiple goroutines
- ability to timeout on a channel send or recv.
- automatic context switching on I/O syscalls.

2.1. Concurrency example - access a shared external resource

```
package main

import "math/rand"

func main() {
    sigs := make([]chan int, 10)
    for i := 0; i < 10; i++ {
        sigs[i] = make(chan int)
        go func(n int, sig chan int) {
            for {
                <-sig
                sig <- n + rand.Int()
            }
        }(i, sigs[i])
    }
    for i := 0; i < 100; i++ {
        sigs[i%10] <- 0
        println("got resource result ", <-sigs[i%10])
    }
}
```

Run

2.2. Concurrency example - timeout

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch := make(chan string)
    go func() {
        time.Sleep(1 * time.Second)
        ch <- "goroutine 1"
    }()
    select {
    case str := <-ch:
        fmt.Println("got string ", str)
    case <-time.After(100 * time.Millisecond):
        fmt.Println("timed out on goroutine1")
    }

}
```

Run